Localization and Centralized Control of a Swarm of Mobile Robots

Adil Faqah

Abstract

In this project I implement an approach for the localization of a swarm of Wi-Fi controlled robots by the aid of fiducial markers. I also successfully implement centralized control with functionalities such as: moving from point A to point B, avoiding obstacles, and showcasing swarm behavior.

1 Introduction

Swarm robotics is a new approach towards multi-robot systems. It involves a large number of, mostly simple, physical robots that can coordinate behavior based upon their interactions between each other and the environment; essentially acting as a group. The primary source of inspiration for Swarm Robot Systems (SRS) is how swarms found in nature, such as social insects, fishes and mammals interact with each other in the swarm in real life. Producing group behaviors that show great flexibility and robustness; including but not limited to: path planning, nest constructing and task allocation.

Perhaps the most major problem encountered when dealing with SRS is the need to determine the position of the robots within the swarm without the use of external references, such as the GPS. The approaches that are normally used to address this problem can be classified as either range-based or range-free method.

In the range-based approach the nodes in the SRS measure their distance to reference nodes or their neighbors using metrics such as Angle of Arrival (AOA), Received Signal Strength Indication (RSSI), Time of Arrival (TOA) and/or Time Difference of Arrival (TDOA). However in the range-free approach, the absolute distance and angle between nodes is not measured. Instead, the location of the nodes is calculated by using network connectivity information. Popular algorithms for the range-free approach are: the centroid algorithm, the amorphous location algorithm and the dv-hop algorithm.

Range-free methods are simpler to implement and more economic than range-based methods. However, their results are often imprecise. On the other hand while range-based methods produce more precise results, they are complex, expensive to implement and may require special equipment one each node (e.g. an antenna array in the case of the AOA method). Hence, in this project I decided to go with the range-free approach based localization, and centralized control.

2 Approach

Before going into details about the actual perception, and control modules. It is important to first go over the different components of the system and how they are related to each other.

The robot platform that is used in the project to create the swarm is an off-the-shelf motorized tank with a webcam, that can be controlled over Wi-Fi by connecting to its Access Point (AP), and using the accompanying Android or iOS app. Fortunately the communication protocol of the robot has already been hacked, which was a huge advantage.

There were however some limitations that needed to be overcome. Firstly, the robot platform only supports 6 individual AA batteries. This was extremely impractical and would be expensive in the long run, as a result each robot was modified, and installed with a 1300 mAh 3 cell LiPo battery along with a 11.1 V to 9 V buck converter.

Secondly, it was impractical that the robot can only be controlled by connecting to its AP. After some investigating it was discovered that the robot platform uses an RTL8196EU chip based Wi-Fi module, which can be reconfigured from AP mode to client mode. Making this change allowed the robot to be connected to a local area Wi-Fi network as a client.

Lastly, the robot platform does not support variable speed motor control. As a result, the only two parameters that can be controlled are the direction, and on/off status. Since these commands are sent over Wi-Fi, it was not possible to implement even software based PWM control.

The rest of the setup consists of a webcam for visual input. The webcam used is the Microsoft LifeCam Studio, which supports 1280 x 720 High Definition (HD) pixel resolution connected through a USB to ethernet and ethernet to USB extender. Additionally, a desktop computer with Intel(R) Core(TM) i3-4130 @ 3.40 GHz x 4 with 8 GB of RAM running Ubuntu version 14.04 LTS, and ROS Indigo was used to run the perception, and control modules. Moreover, a TP-LINK Archer C50 router was used to create a WLAN to connect to the robots and send control info.

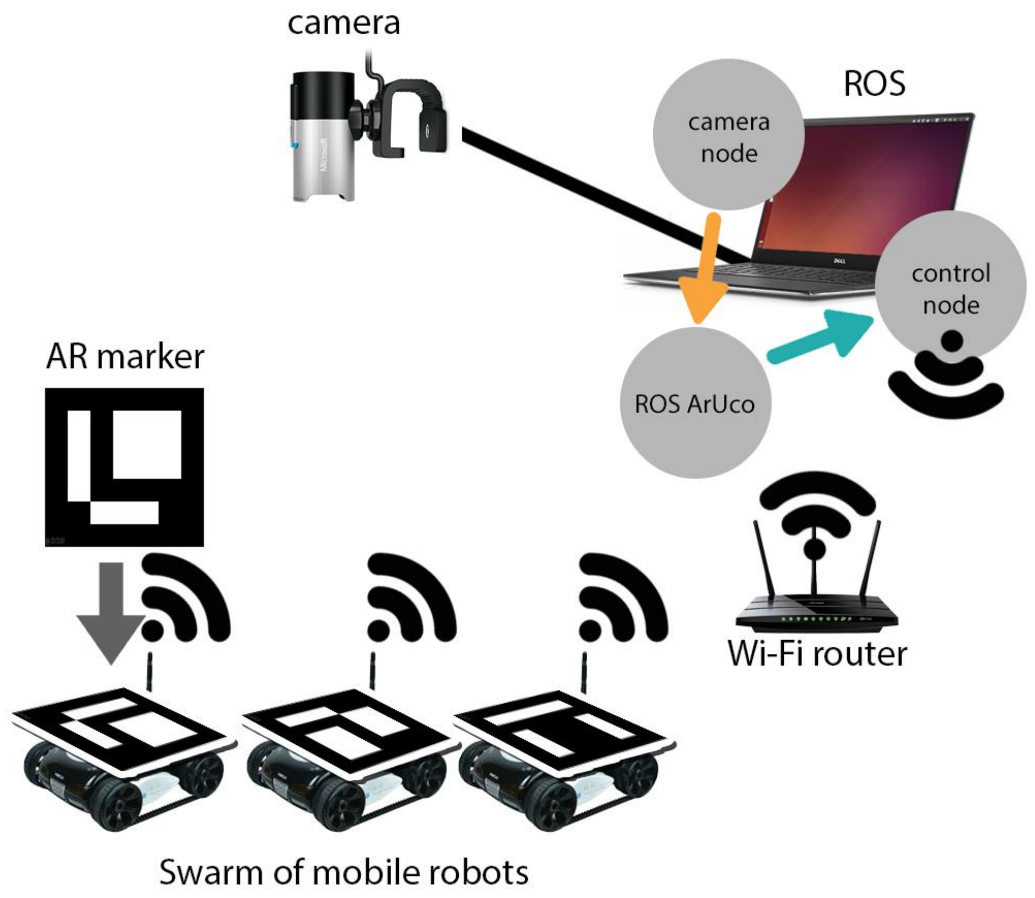


Figure 1: System overview diagram

Figure 1. shows how all these system components are connected to each other. The camera node captures frames through the webcam and publishes them to the camera topic. The ArUCo node (that is the perception module) detects and localizes the marker positions, and then publishes them to a ROS topic. The control node subscribes to that topic and performs the decision making, path planning, and sends out the control signals over Wi-Fi to the mobile robots. These components (sans the computer and the Wi-Fi router) can be seen in Figure 2. A closer look at one of the mobile robots with an AR marker mounter on it can be seen in Figure 3.

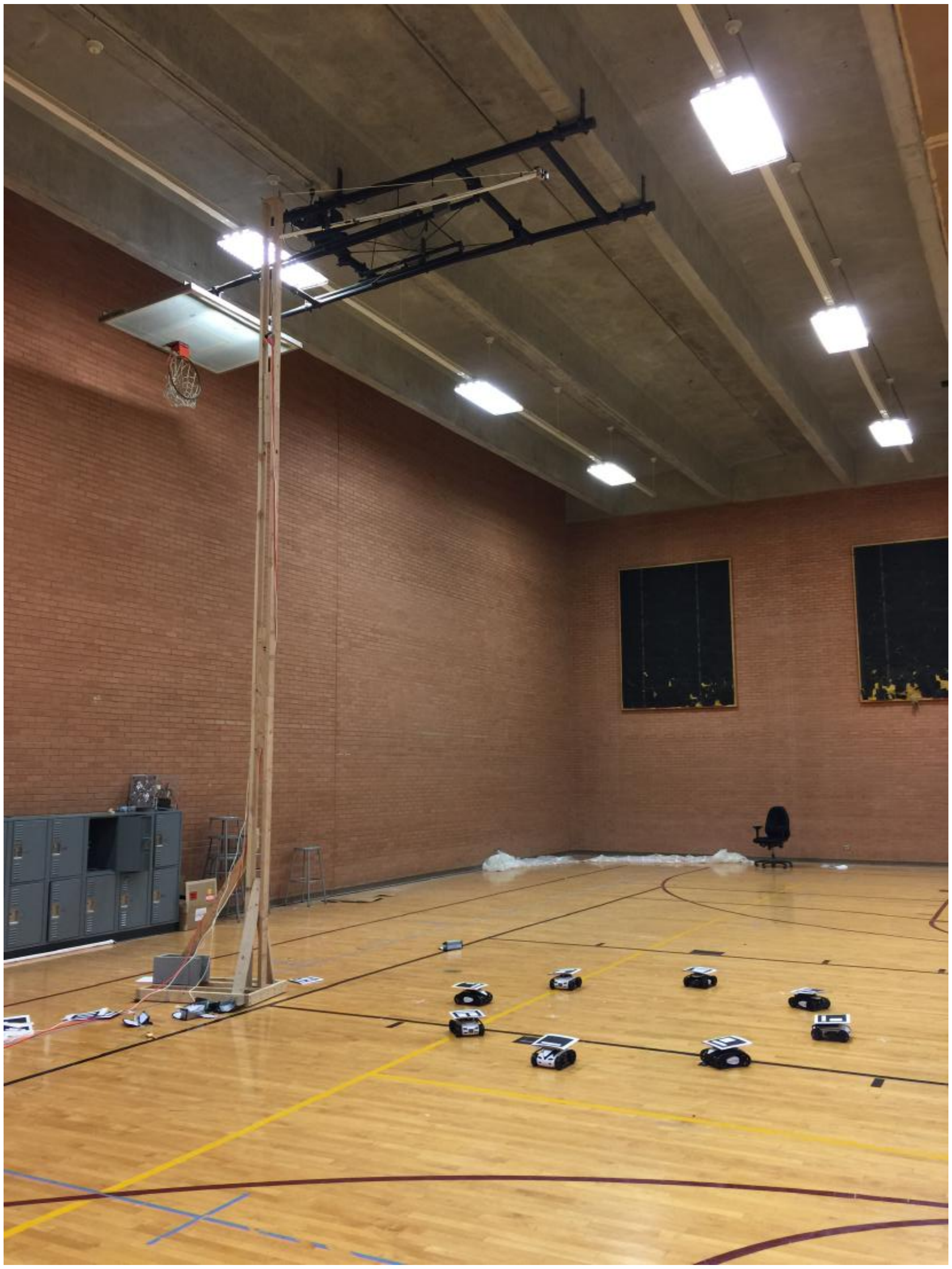


Figure 2: A picture showing the webcam mounted on a tall, vertical structure and a swarm of mobile robots underneath

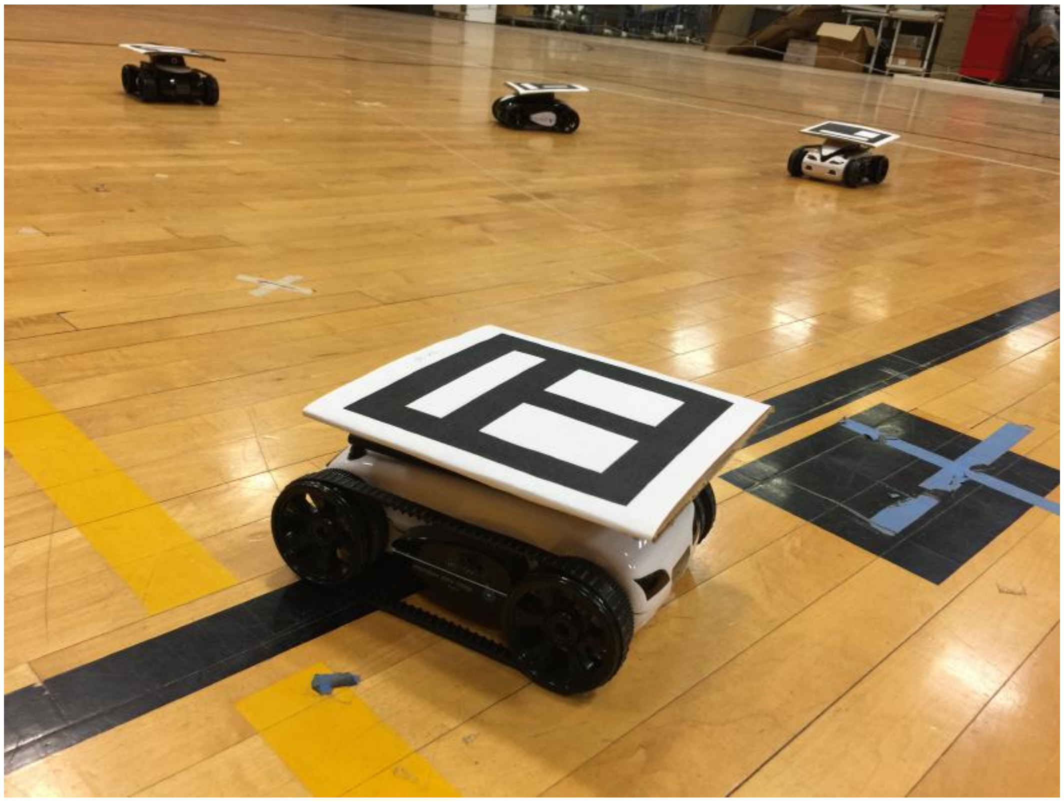


Figure 3: One of the mobile robots with an AR marker mounted on top of it

2.1 Perception Module

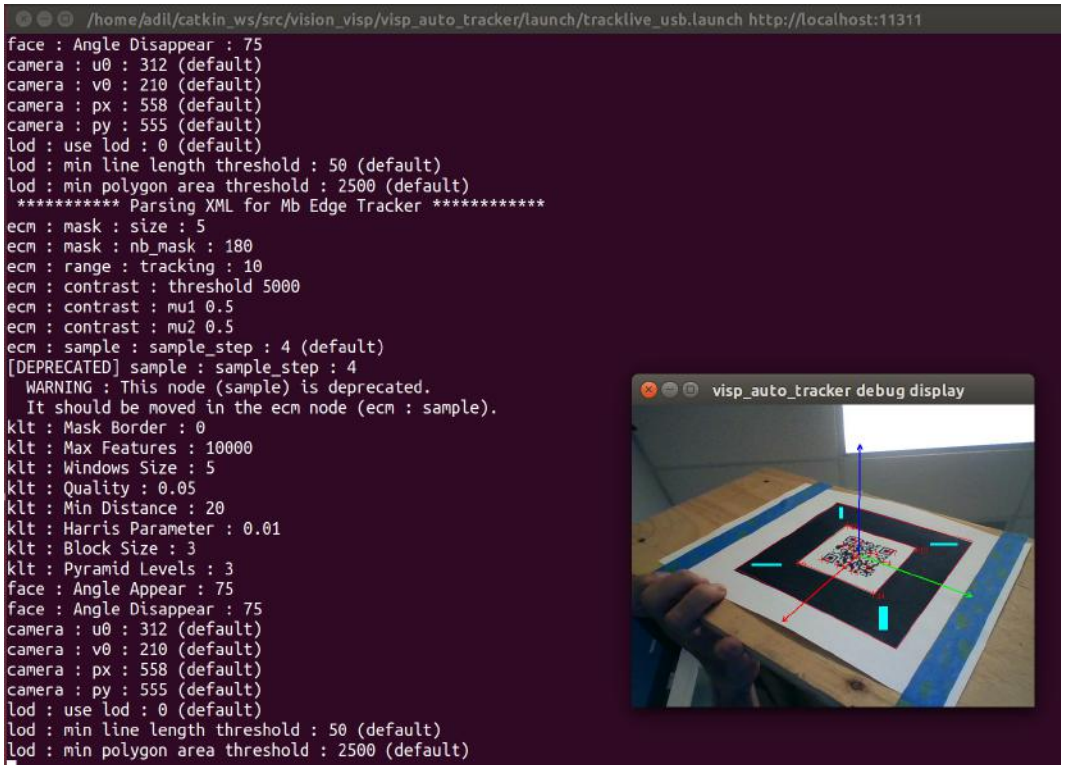
While there are a variety of methods for position and pose tracking, for this project I needed a solution that would be fast as well as robust. Hence, the best approach was to use fiducial markers. Instead of re-inventing the wheel I decided to explore solutions that were already developed.

The first library that I examined was the visp_auto_tracker library (seen in Figure 4). However while testing it I found that while it was very robust, the frame rate was not very high. More importantly, it was unable to track the markers if they were at a distance from the camera since it tracked only QR markers which, unfortunately, can have very complex designs. This can be seen in Figure 5 where a QR marker is practically indistinguishable, next to an AR marker.

I eventually came across the ArUco library. A minimal library primarily designed for Augmented Reality applications based on OpenCV. Initial tests showed that the ArUco library was fast and robust. And since the markers are very simple, they can be easily and reliably tracked even at a distance.

Unfortunately, I was not able to find a working version of ArUco in ROS, hence, I had to port it myself. I wrote a ROS node in C++ which uses the ArUco library to detect all markers in a given frame (which it obtains from the camera node). Once detected, the pose and position is estimated by the help of the camera parameters which are obtained from the camera calibration file. The only desired parameters are the x position, y position and the rotation around the z axis. However since the ArUco library returns the rotations in quaternions I had to convert them to euclidean angles first. Finally, the marker id, x position, y position, theta and time stamp for each marker are published to a ROS topic called `chatter` with the data type `Float64MultiArray`. The node then moves on to the next frame.

An important step for the perception node was to perform camera calibration. This involved taking images of a ChArUco board (show in Figure 6) at different orientations and distances, and then passing them through a command line calibration utility. The calibration utility output a .yml



```
/home/adil/catkin_ws/src/vision_visp/visp_auto_tracker/launch/tracklive_usb.launch http://localhost:11311
face : Angle Disappear : 75
camera : u0 : 312 (default)
camera : v0 : 210 (default)
camera : px : 558 (default)
camera : py : 555 (default)
lod : use lod : 0 (default)
lod : min line length threshold : 50 (default)
lod : min polygon area threshold : 2500 (default)
 *********** Parsing XML for Mb Edge Tracker ************
ecm : mask : size : 5
ecm : mask : nb_mask : 180
ecm : range : tracking : 10
ecm : contrast : threshold 5000
ecm : contrast : mu1 0.5
ecm : contrast : mu2 0.5
ecm : sample : sample_step : 4 (default)
[DEPRECATED] sample : sample_step : 4
  WARNING : This node (sample) is deprecated.
  It should be moved in the ecm node (ecm : sample).
klt : Mask Border : 0
klt : Max Features : 10000
klt : Windows Size : 5
klt : Quality : 0.05
klt : Min Distance : 20
klt : Harris Parameter : 0.01
klt : Block Size : 3
klt : Pyramid Levels : 3
face : Angle Appear : 75
face : Angle Disappear : 75
camera : u0 : 312 (default)
camera : v0 : 210 (default)
camera : px : 558 (default)
camera : py : 555 (default)
lod : use lod : 0 (default)
lod : min line length threshold : 50 (default)
lod : min polygon area threshold : 2500 (default)
```

Figure 4: Tracking a QR marker in ROS using the visp_auto_tracker

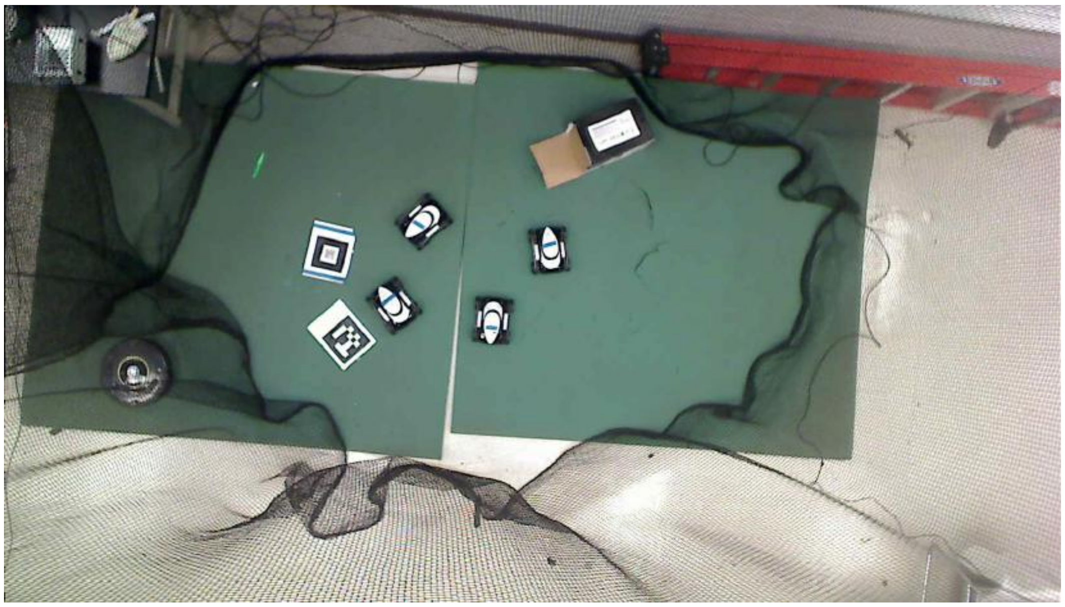


Figure 5: A top down view of the initial test setup

file consisting of information about the camera intrinsics, and the distortion matrix. This was vital in ensuring that the pose and position tracking was accurate.

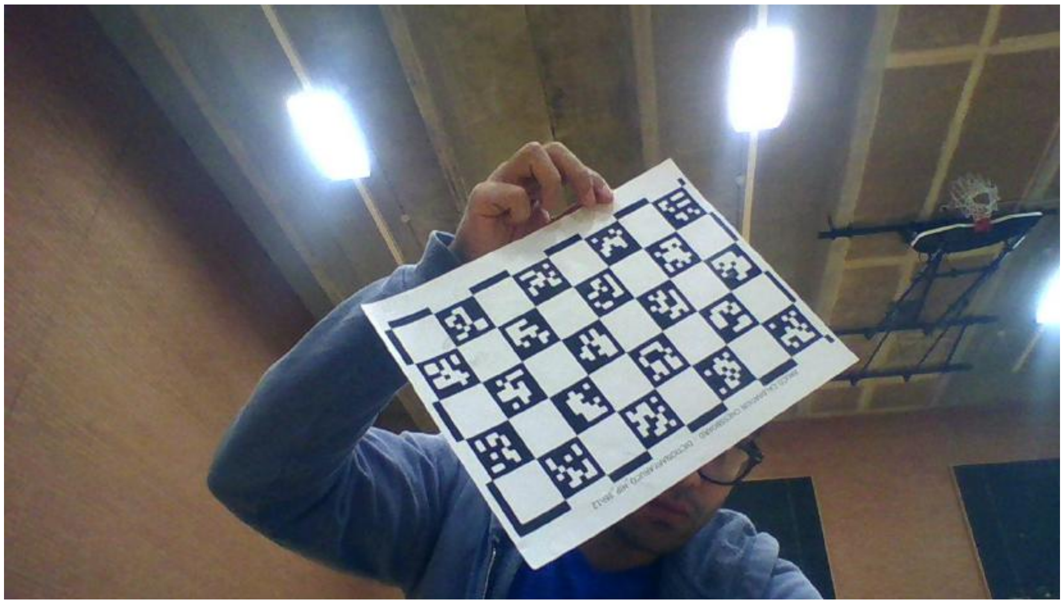


Figure 6: A top down view of the initial test setup

2.2 Control Module

The control module, i.e. the control node is written in Python 2.7 using OOP. It is responsible for using the localization data to make decisions, perform path planning, and send the necessary control commands.

The robots are controlled through a Transmission Control Protocol (TCP) connection over port 8150. Each robot has been assigned a fixed IP address through MAC binding.

During the initialization sequence, connection is established with each robot in the swarm through sockets. A new `MobileRobot` object is created, and initialized for each robot. The identifiers are maintained in a dictionary.

Methods related to controlling movement are `move_forward`, `move_backward`, `turn_left`, `turn_right` and `stop`.

The control node subscribes to the `chatter` topic with the data type `Float64MultiArray`. Whenever new localization data is published by the perception node, a callback function is called. For each marker, the callback function unpacks the localization data, adds it to the global obstacle list, and updates the current position of the marker associated with that robot. Finally it sets and clears an event flag to signal the mobile robot thread that the position data has been updated.

2.2.1 Moving from current position to desired position

The process of moving from the current position and orientation of the robot to a given desired position and orientation involves the following steps, in essentially an infinite loop:

1. **The robot uses proportional control, and rotates until it is facing the x and y coordinates of the desired position**
2. **The robot performs obstacle detection on the line segment between its initial and desired position**. This can result in the following three scenarios:
   (a) **The path is clear and the robot can move forward**
   The robot computes its euclidean distance to the desired position. If the distance computed is outside the acceptable translation error, the robot starts moving forward

and the control flow goes to **1**. Else, it stops and breaks out of the loop, indicating that the robot has arrived at its target position.

If the position is reached is an intermediate desired position (i.e. a detour waypoint), the control flow will go to **1**. However, if the position reached is the actual desired position, the control flow will go to **3**.

(b) **The robot is not near its desired position and there is an obstacle in the way**
In this case a detour waypoint is computed. The detour waypoint is computed on the normal of the line between the robot's current position and the desired position. Depending on which side of the line the center of the obstacle lies on, the normal on the opposite side of the line is picked. This ensures that the detour waypoint lies on the shortest path.

Once the detour waypoint is computed, the robot's intermediate desired position is updated, a nested move command is called, and the control flow essentially goes to **1**. Note, that since multiple obstacles may lie on the line segment, only the closest obstacle is considered.

(c) **The robot is close to its actual desired position, however there is an obstacle blocking the desired position**
In this case no detour waypoint is computed. The robot essentially stops and periodically checks for the desired position to clear. If and when that happens, the control flow goes to **2 (a)**.

3. **The robot rotates until it matches the desired orientation**

2.2.2 Demonstrating swarming behavior

The swarming behavior implemented in the project involves the robots being arranging themselves in a certain formation. This process can effectively be divided into three steps.

1. **Generate a list of points N (where N is the number of robots in the swarm) for a certain position**
For this project, I wrote a function that takes in the x and y co-ordinates of the center of a circle, along with the radius of that circle, and the number of robots in the swarm. It then uses the parametric equations for a circle to return a list of N equidistant points (x, y, θ) that lie on that circle.
2. **Plan the formation**
This involves a generic algorithm that was also implemented as a function. The function takes in a list of points N (x, y, θ) and a list of positions (x, y, θ) of N robots. For each point it computes the distance of that point with each robot in the list. Finally, it iterates through the list of points, and for each point it assigns it the robot closes to it and then removes the robot from the list of available robots. In the end it returns a list of N robot_id:point pairs.
3. **Update the desired position for each robot depending on the assignment by the formation planner**
In this step, the formation plan is used to start a thread for each mobile robot in the swarm and update its target position.

3 Experiments

3.1 Data Collection

The first step to ensure accurate tracking and pose estimation was to perform calibration.

In order to make the testing and debugging of the control algorithms easier, the control node was made to output detailed log files of the following format:

```
    <time stamp>, <higher level function>, <target parameter for that
higher level function>, <lower level function>, <current x position>,
<current y position>, <current theta position>
```

For example:

```
    02:15:57.637, rotate_to_xy, target_angle = 141.0, turn_right, x = 285,
y = 124, theta = -13
```

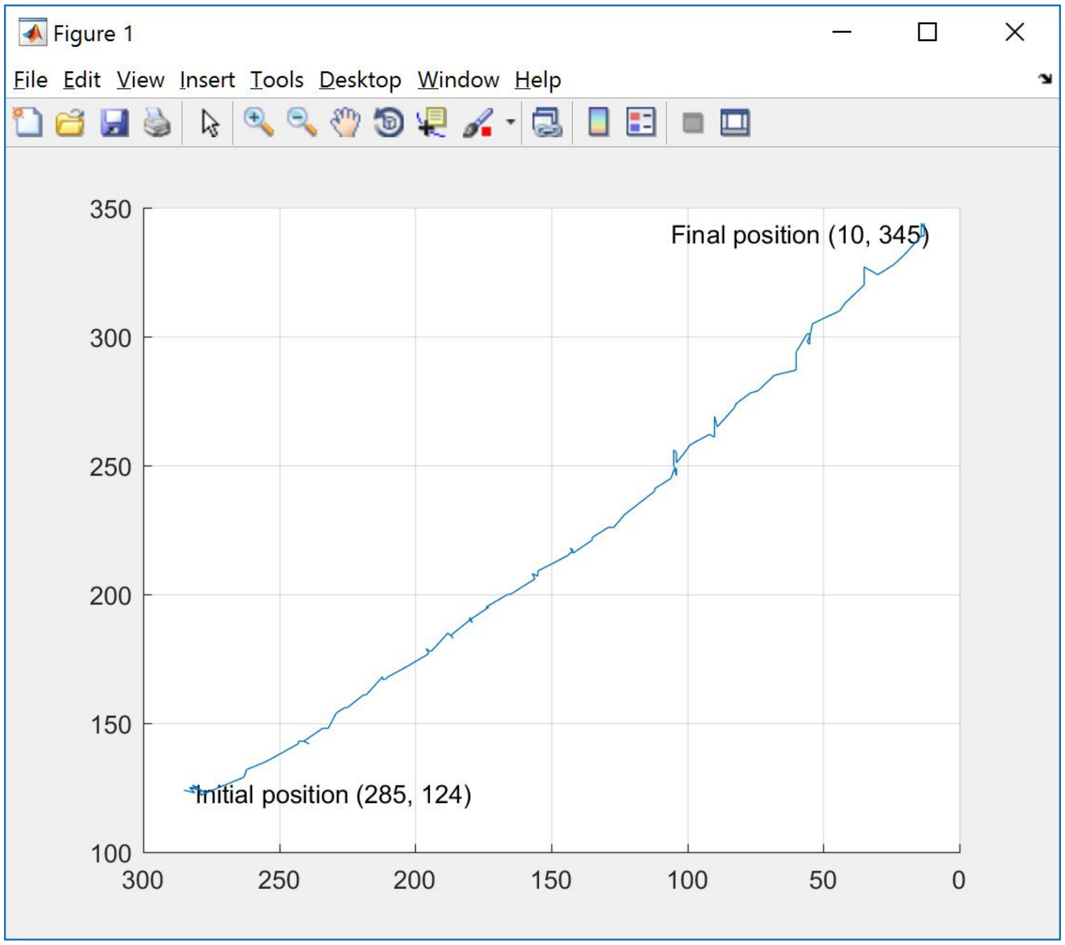


Figure 7: A plot of the path followed by a robot generated using the log file

These log files were used to generate graphs in MATLAB such as Figure 7. The log files, in conjunction with the plots allowed for fast identification and resolution of hard to find bugs.

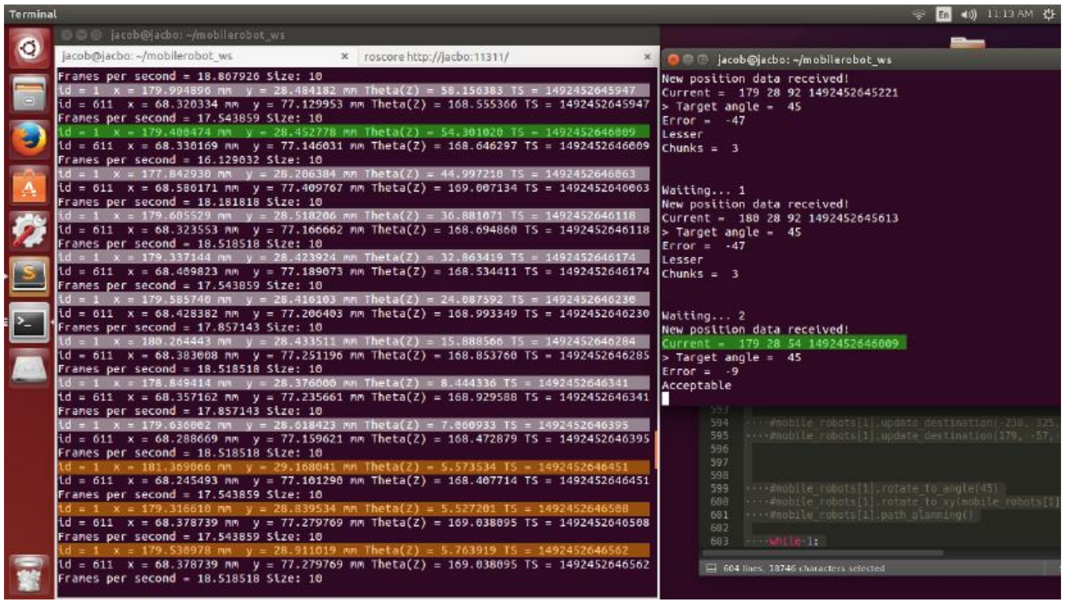
Another methodology was to output detailed information on to the command line output.

Additionally, the following experiments were performed in order to sequentially test all the functionality that was implemented:

1. Moving a robot from initial position to desired position and orientation
2. Moving to a desired position and orientation while avoiding a static obstacle
3. Moving to a desired position whole avoiding a moving obstacle
4. Moving to a desired position that is obstructed
5. Simultaneously controlling 4 robots
6. Controlling a swarm of 8 robots to create a formation

3.2 ROS System Setup

For the ROS system setup, a desktop computer with Intel(R) Core(TM) i3-4130 @ 3.40 GHz x 4 with 8 GB of RAM running Ubuntu version 14.04 LTS, and ROS Indigo was used. There are three main nodes:



```
Frames per second = 18.867926 Size: 10
id = 1  x = 179.994896 mm  y = 28.484182 mm Theta(Z) = 58.156383 TS = 1492452645947
id = 611  x = 68.320334 mm  y = 77.129953 mm Theta(Z) = 168.555366 TS = 1492452645947
Frames per second = 17.543859 Size: 10
id = 1  x = 179.400474 mm  y = 28.452778 mm Theta(Z) = 54.301020 TS = 1492452646009
id = 611  x = 68.330169 mm  y = 77.146031 mm Theta(Z) = 168.646297 TS = 1492452646009
Frames per second = 16.129032 Size: 10
id = 1  x = 177.842930 mm  y = 28.206384 mm Theta(Z) = 44.997210 TS = 1492452646063
id = 611  x = 68.586171 mm  y = 77.409767 mm Theta(Z) = 169.007134 TS = 1492452646063
Frames per second = 18.181818 Size: 10
id = 1  x = 179.605529 mm  y = 28.518206 mm Theta(Z) = 36.881071 TS = 1492452646118
id = 611  x = 68.323553 mm  y = 77.166662 mm Theta(Z) = 168.694860 TS = 1492452646118
Frames per second = 18.518518 Size: 10
id = 1  x = 179.337144 mm  y = 28.423924 mm Theta(Z) = 32.863419 TS = 1492452646174
id = 611  x = 68.409823 mm  y = 77.189073 mm Theta(Z) = 168.534411 TS = 1492452646174
Frames per second = 17.543859 Size: 10
id = 1  x = 179.585740 mm  y = 28.416103 mm Theta(Z) = 24.087592 TS = 1492452646230
id = 611  x = 68.428382 mm  y = 77.206403 mm Theta(Z) = 168.993349 TS = 1492452646230
Frames per second = 17.857143 Size: 10
id = 1  x = 180.264443 mm  y = 28.433511 mm Theta(Z) = 15.888566 TS = 1492452646284
id = 611  x = 68.383008 mm  y = 77.251196 mm Theta(Z) = 168.853760 TS = 1492452646285
Frames per second = 18.518518 Size: 10
id = 1  x = 178.849414 mm  y = 28.376000 mm Theta(Z) = 8.444336 TS = 1492452646341
id = 611  x = 68.357162 mm  y = 77.235661 mm Theta(Z) = 168.929588 TS = 1492452646341
Frames per second = 17.857143 Size: 10
id = 1  x = 179.636002 mm  y = 28.618423 mm Theta(Z) = 7.000933 TS = 1492452646395
id = 611  x = 68.288669 mm  y = 77.159621 mm Theta(Z) = 168.472879 TS = 1492452646395
Frames per second = 18.518518 Size: 10
id = 1  x = 181.369066 mm  y = 29.168041 mm Theta(Z) = 5.573534 TS = 1492452646451
id = 611  x = 68.245493 mm  y = 77.101290 mm Theta(Z) = 168.407714 TS = 1492452646451
Frames per second = 17.543859 Size: 10
id = 1  x = 179.316610 mm  y = 28.839534 mm Theta(Z) = 5.527201 TS = 1492452646508
id = 611  x = 68.378739 mm  y = 77.279769 mm Theta(Z) = 169.038095 TS = 1492452646508
Frames per second = 17.543859 Size: 10
id = 1  x = 179.530978 mm  y = 28.911019 mm Theta(Z) = 5.763919 TS = 1492452646562
id = 611  x = 68.378739 mm  y = 77.279769 mm Theta(Z) = 169.038095 TS = 1492452646562
Frames per second = 18.518518 Size: 10
```

```
New position data received!
Current =  179 28 92 1492452645221
> Target angle =  45
Error =  -47
Lesser
Chunks =  3


Waiting... 1
New position data received!
Current =  180 28 92 1492452645613
> Target angle =  45
Error =  -47
Lesser
Chunks =  3


Waiting... 2
New position data received!
Current =  179 28 54 1492452646009
> Target angle =  45
Error =  -9
Acceptable
```

```
    #mobile_robots[1].update_destination(-230, 325,
    #mobile_robots[1].update_destination(179, -57,



    #mobile_robots[1].rotate_to_angle(45)
    #mobile_robots[1].rotate_to_xy(mobile_robots[1]
    #mobile_robots[1].path_planning()

    while 1:
```

Terminal
jacob@jacbo: ~/mobilerobot_ws
roscore http://jacbo:11311/
11:13 AM
604 lines, 18746 characters selected

Figure 8: A screenshot showing detailed debugging information being output to the command line

1. usb_cam: The camera node is responsible for fetching frames from the camera
2. aruco_simple: This node reads the frames from the usb_cam node, detects markers, creates a vector of marker ids, positions, orientations, and time stamps and publishes them to a ROS topic called `chatter`
3. control_node: This node reads the localization data from the `chatter` topic, and performs decision making, path planning, and sends out the control signals.
4.

3.3 Experimental Results

All the tests were executed successfully and their results are outlined below:

3.3.1 Moving a robot from initial position to desired position and orientation

3.3.2 Moving to a desired position and orientation while avoiding a static obstacle

3.3.3 Moving to a desired position whole avoiding a moving obstacle

3.3.4 Moving to a desired position that is obstructed

3.3.5 Simultaneously controlling 4 robots

3.3.6 Controlling a swarm of 8 robots to create a formation

4 Discussion and Conclusion

In this project I was able to successfully write a perception node integrating the ArUco library into ROS in order to localize a swarm of up to eight robots. Moreover, I was successful in writing a control node that is capable of simultaneous controlling multiple robots (up to 8 tested). I implemented functionality such as moving from one point to another, avoiding both stationary and non-stationary obstacles, as well as dealing with situations where the destination position may be blocked. Lastly, I was successful in efficiently implementing and executing swarming behavior.

While I was successful in implementing and achieving what I set out to accomplish at the start of this project, there are certainly some aspects that can be developed further. For instance, imple-

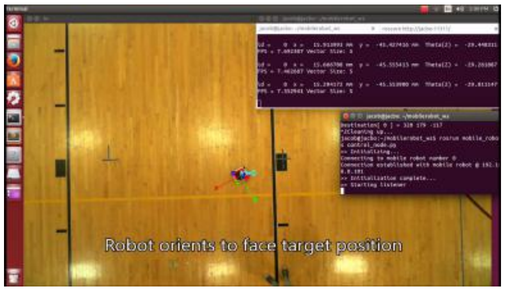


Figure 9: The robot starts with an initial position and orientation

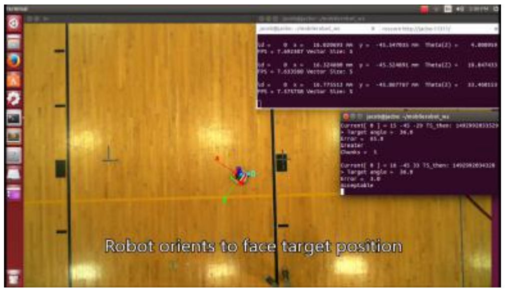


Figure 10: The robot orients itself to face the desired position

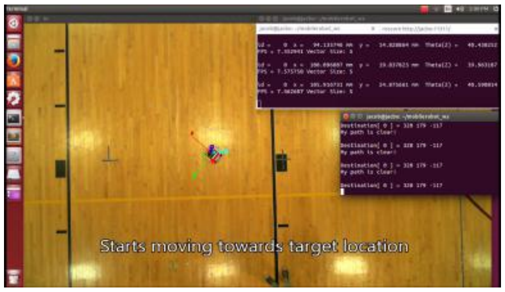


Figure 11: The robot starts moving towards the desired position

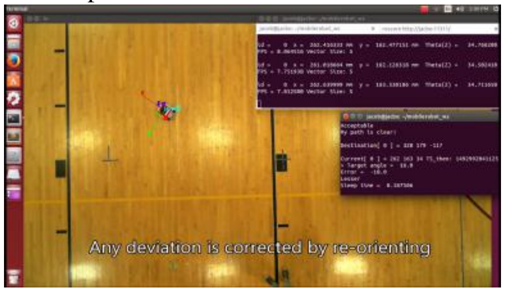


Figure 12: The robot corrects any deviations by re-orienting

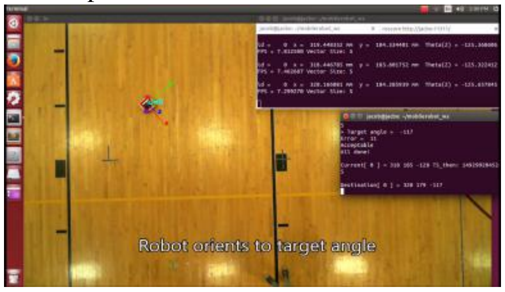


Figure 13: At the desired position, the robot rotates to target position

Figure 14: Moving a robot from initial position to desired position and orientation

menting Bezièr curve based path planning and obstacle avoidance for more robust and smoother movement and performance.

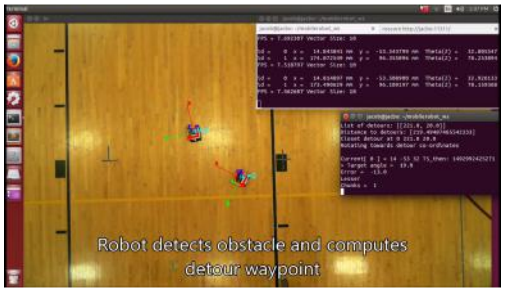


Figure 15: The robot detects an obstacle in its path and computes a detour waypoint

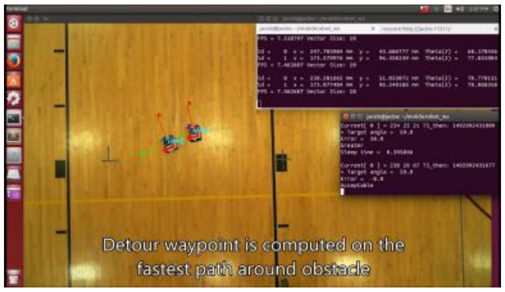

Detour waypoint is computed on the fastest path around obstacle

Figure 16: The detour waypoint is computed on the fastest path

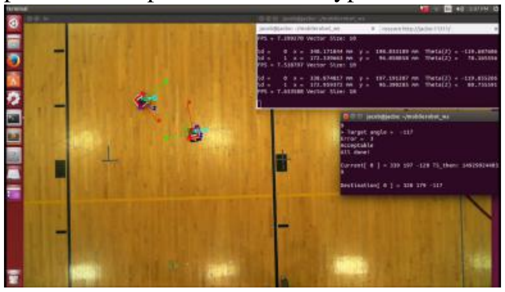


Figure 17: The robot arrives at the desired position

Figure 18: Moving to a desired position and orientation while avoiding a static obstacle

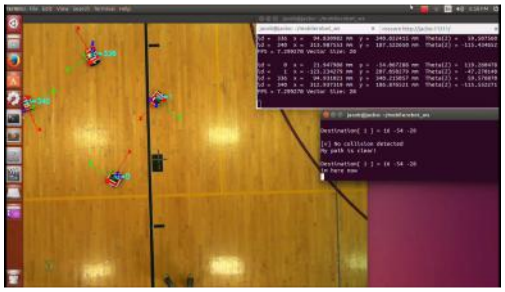


Figure 19: Each robot moves towards the other robot's initial position

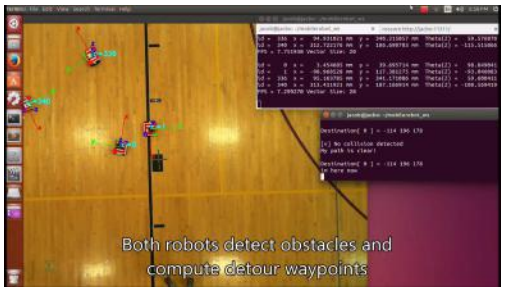


Figure 20: Both robots detect each other as obstacles and computer their respective detour waypoints

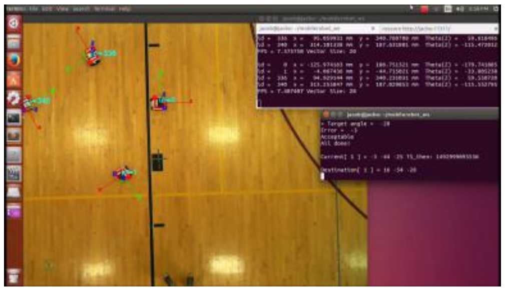


Figure 21: Both robots arrive at their final position

Figure 22: Moving to a desired position whole avoiding a moving obstacle

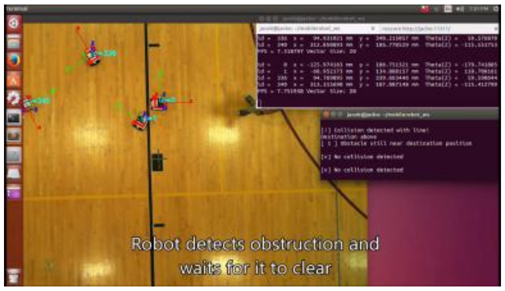


Figure 23: The robot detects an obstruction and waits for it to clear

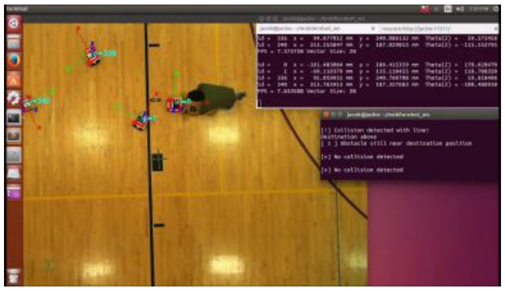


Figure 24: The robot observes that the obstruction is removed

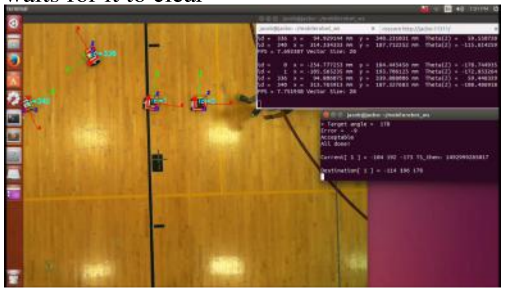


Figure 25: The robot moves to the desired position

Figure 26: Moving to a desired position that is obstructed

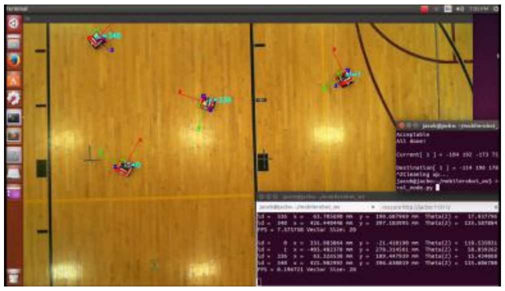


Figure 27: The robots start with random positions and orientations

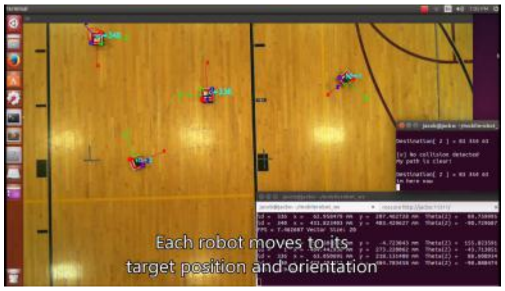


Figure 28: Each robot is assigned a target position and orientation

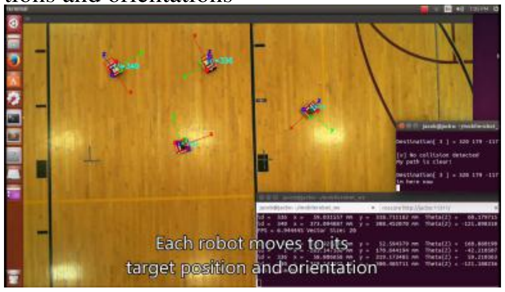


Figure 29: Each robot moves to its target position and orientation

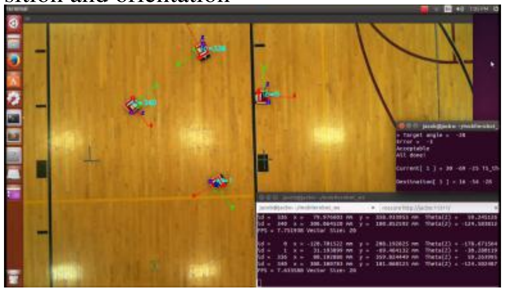


Figure 30: The robots arrive at their target position

Figure 31: Simultaneously controlling 4 robots

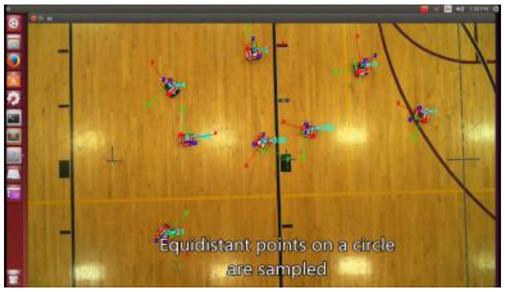


Figure 32: Equidistant points on a circle are sampled

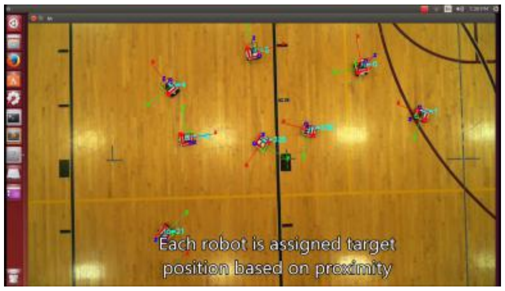


Figure 33: Each robot is assigned to a point that it is closest to

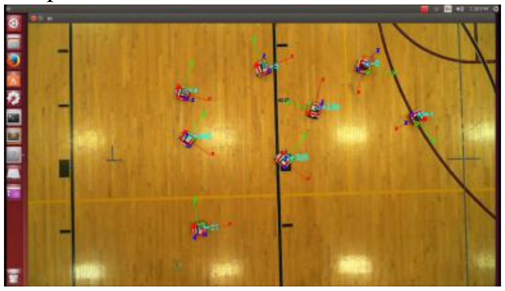


Figure 34: The desired position for each robot is updated

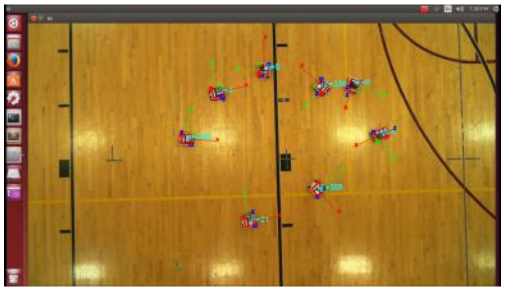


Figure 35: The robots move towards their desired position to create the formation

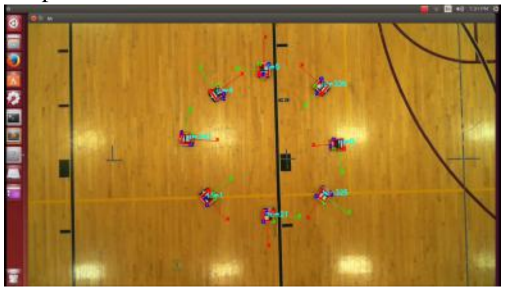


Figure 36: The robots complete the formation

Figure 37: Controlling a swarm of 8 robots to create a formation